1.3 Representing Data in Python

Data is all around us, but so are computers. If decisions must be data-driven then computers are an excellent tool for processing that data. Especially when we consider that computers are several orders of magnitude faster at computing data than a human. The problem is that computers need to be told exactly *how* to process the data, and we can do so using one of several programming languages. In this section, we see how data types are represented in Python and how we can use Python to perform operations for us. We'll learn about some subtle, but crucial, differences between our theoretical definitions of data types from Section 1.1 and what Python can actually represent. But first, we'll introduce some general terminology for using the interactive Python console.

The Python console

When we first start the Python console, we see the following:

```
>>>
```

The text `>>>` is called the Python **prompt**: the console is "prompting" us to type in some Python code to execute. If we type in a simple arithmetic expression,

```
>>> 4 + 5
```

and press Enter, we see the following output:

```
>>> 4 + 5
9
```

The interpreter took our bit of code, `4 + 5`, and calculated its value, `9`. A piece of Python code that produces a value is called an **expression**, and the act of calculating the value of an expression is called **evaluating** the expression.

The expression `4 + 5` looks simple enough, but technically it is formed from two smaller expressions—the numbers `4` and `5` themselves. We can ask `Python` to evaluate each of these, though the result is not very interesting.

```
>>> 4
4
```

```
>>> 5
5
```

A Python **literal** is the simplest kind of Python expression: it is a piece of code that represents the exact value as written. For example, `4` is an integer literal representing the number 4.

To sum up,[1] the expression `4 + 5` consists of two smaller expressions, the literals `4` and `5`,

[1] The pun was not originally intended, but we are pointing it out…

joined together with the arithmetic operator `+`, representing addition. We'll devote the rest of this section to exploring the different kinds of data types we can use in Python: both how to write their literals, and what operations we can perform on them.

Numeric data in Python (`int`, `float`)

Python has two data types for representing numeric data: `int` and `float`. Let's start with `int`, which stands for "integer", and is the data type that Python uses to represent integers.

An `int` literal is simply the number as a sequence of digits with an optional `-` sign, like `110` or `-3421`.

Python supports all of the arithmetic operations we discussed in Section 1.1. Here are some examples; try typing them into the Python console yourself to follow along!

```
>>> 2 + 3
5
>>> 2 - 5
-3
>>> -2 * 10
-20
>>> 2 ** 5  # This is "2 to the power of 5"
32
```

In the last prompt, we included some additional text—`# This is "2 to the power of 5"`. In Python, we use the character `#` in code to begin a **comment**, which is code that is ignored by the Python interpreter. Comments are only meant for humans to read, and are a useful way of providing additional information about some Python code. We used it above to explain the meaning of the `**` operator in our demo.

Python supports the standard precedence rules for arithmetic operations,[2] performing

[2] sometimes referred to as "BEDMAS" or "PEMDAS"

exponentiation before multiplication, and multiplication before addition and subtraction:

```
>>> 1 + 2 ** 3 * 5  # Equal to "1 plus ((2 to the power of 3) times 5)"
41
```

Just like in mathematics, long expressions like this one can be hard to read. So Python also allows you to use parentheses to group expressions together:

```
>>> 1 + ((2 ** 3) * 5)  # Equivalent to the previous expression
41
>>> (1 + 2) ** (3 * 5)  # Different grouping: "(1 plus 2) to the power of (3
        times 5)"
14348907
```

When we add, subtract, multiply, and use exponentiation on two integers, the result is always an integer, and so Python always produces an `int` value for these operations. But *dividing* two integers certainly doesn't always produce an integer—what does Python do in this case? It turns out that Python has two different forms of division. The first is the operator `//`, and is called **floor division** (or sometimes **integer division**). For two integers `x` and `y`, the result of `x // y` is equal to the quotient $\frac{x}{y}$, rounded down to the nearest integer. Here are some examples:

```
>>> 6 // 2
3
>>> 15 // 2  # 15 ÷ 2 = 7.5, and // rounds down
7
>>> -15 // 2  # Careful! -15 ÷ 2 = -7.5, which rounds down to -8
-8
```

But what about "real" division? This is done using the division operator `/`:

```
>>> 5 / 2
2.5
```

This returns `2.5`, which is a value of Python's `float` type, which Python uses to represent arbitrary real numbers. A `float` literal is written as a sequence of digits followed by a decimal point (`.`) and then another sequence of digits. `2.5`, `.123`, and `1000.00000001` are all examples of `float` literals.

All of the arithemtic operations we've looked at so far work with `float` values too:

```
>>> 3.5 + 2.4
5.9
>>> 3.5 - 20.9
-17.4
```

```
>>> 3.5 * 2.5
8.75
>>> 3.5 / 2.5
1.4
>>> 2 ** 0.5
1.4142135623730951
```

The last expression, `2 ** 0.5`, calculates the square root of 2. However, this actually poses a problem for Python: since $\sqrt{2}$ is an irrational number, its decimal expansion is infinite, and so it cannot be represented in any finite amount of computer memory.[3]

[3] More precisely, computers use a binary system where all data, including numbers, are represented as a sequence of 0s and 1s. This sequence of 0s and 1s is finite since computer memory is finite, and so cannot exactly represent $\sqrt{2}$. We will discuss this binary representation of numbers later this year.

] The `float` value that's produced, `1.4142135623730951`, is an approximation of $\sqrt{2}$, but is not equal to it. Let's see what happens if we try to square it:

```
>>> 1.4142135623730951 * 1.4142135623730951
2.0000000000000004
```

This illustrates a fundamental limitation of `float`: this data type is used to represent real numbers, but cannot always represent them exactly. Rather, a `float` value *approximates* the value of the real number; sometimes that approximation is exact, like `2.5`, but most of the time it isn't.

3 vs. 3.0

Here's an oddity:

```
>>> 6 // 2
3
>>> 6 / 2
3.0
```

Even though $\frac{6}{2}$ is mathematically an integer, the results of the division using `//` and `/` are subtly different in Python. When `x` and `y` are `int`s, `x // y` *always* evaluates to an `int`, and `x / y` *always* evaluates to a `float`, even if the value of $\frac{x}{y}$ is an integer! So `6 // 2` has value `3`, but `6 / 2` has value `3.0`. These two values represent the same mathematical quantity—the number 3—but are stored as different data types in Python, something we'll explore more later in this course when we study how `int`s and `float`s actually work in Python.

Mixing `int`s and `float`s

So to summarize: for two `int`s `x` and `y`, `x + y`, `x - y`, `x * y`, `x // y`, and `x ** y` all produce `int`s, and `x / y` always produces a `float`. For two `float`s, it's even simpler: all six of these arithmetic operations produce a `float`.[4]

[4] Even `//`. Try it!

But what happens when we mix these two types? *An arithmetic operation that is given one `int` and one `float` always produces a `float`.* You can think of a `float` as a parasite—even in long arithmetic expressions where only one value is a `float`, the whole expression will evaluate to a `float`.

```
>>> 12 - 4 * 5 // (3.0 ** 2) + 100
110.0
```

Operation	Description
`a + b`	Returns the sum of the `a` and `b`
`a - b`	Returns the result of subtraction of `b` from `a`
`a * b`	Returns the result of multiplying `a` by `b`
`a / b`	Return the result of dividing `a` by `b`
`a % b`	Return the remainder when `a` is divided by `b`
`a ** b`	Return the result of `a` being raised to the power of `b`
`a // b`	Return the floored division `a / b`

Comparison operators

Finally, the numeric comparison operators are represented in Python as follows:

Operation	Description
`a == b`	Returns whether `a` and `b` are equal.
`a != b`	Returns whether `a` and `b` are *not* equal (opposite of `==`).
`a > b`	Returns whether `a` is greater than the value of `b`.
`a < b`	Returns whether `a` is less than the value of `b`.
`a >= b`	Returns whether `a` is greater than or equal to `b`.
`a <= b`	Returns whether `a` is less than or equal to the value `b`.

Here are a few examples:

```
>>> 4 == 4
True
>>> 4 != 6
True
>>> 4 < 2
False
```

```
>>> 4 >= 1
True
```

And returning to our discussion earlier, we can see that even though `int`s and `float`s are different types, Python can recognize when their values represent the exact same number:

```
>>> 4 == 4.0
True
```

In these examples, we've seen the values `True` and `False` produced as a result of these comparison expressions. You can probably tell exactly what they mean, but let's take a moment to introduce them formally.

Boolean data in Python (`bool`)

In Python, boolean data is represented using the data type `bool`. Unlike the broad range of numbers we just saw, there are only two literal values of type `bool`: `True` and `False`.

There are three boolean operators we can perform on boolean values: `not`, `and`, and `or`.

```
>>> not True
False
>>> True and True
True
>>> True and False
False
>>> False or True
True
>>> False or False
False
```

One note about the `or` operator in Python is that it is the **inclusive or**, meaning it produces `True` when both of its operand expressions are `True`.

```
>>> True or True
True
```

Just as we saw how arithmetic operator expressions can be nested within each other, we can combine boolean operator expressions, and even the arithmetic comparison operators:

```
>>> True and (False or True)
True
```

```
>>> (3 == 4) or (5 > 10)
False
```

Textual data in Python (`str`)

All Python code is text that we type into the computer, so how do we distinguish between text that's code and text that's data, like a person's name? Python uses the `str` (short for "string") data type to represent textual data. A `str` literal is a sequence of characters surrounded by single-quotes (`'`).[5] For example, we could write this course's name in Python as the string literal `'Foundations of Computer Science I'`.

[5] Python allows string literals to be written using either single-quotes or double-qutoes (`"`). We'll tend to use single-quotes in this course to match how Python displays strings, as we'll see in this section.

Now let's see what kinds of operations we can perform on strings. First, we can compare strings using `==`, just like we can for `int`s and `float`s:

```
>>> 'David' == 'David'
True
>>> 'David' == 'david'  # String comparisons are case-sensitive
False
```

Python supports **string indexing** to extract a single character from a string.[6]

[6] Remember, string indexing starts at 0. `s[0]` represents the *first* character in the string `s`.

```
>>> 'David'[0]
'D'
>>> 'David'[3]
'i'
```

And Python supports **concatenation** using the familiar `+` operator:

```
>>> 'One string' + 'to rule them all.'
'One stringto rule them all.'
>>> 'One string ' + 'to rule them all.'  # Note the extra space!
'One string to rule them all.'
```

One operation that we did not cover in Section 1.1 is a fun quirk of Python: string repetition.

```
>>> 'David' * 3
'DavidDavidDavid'
```

And of course, all of these string operation expressions can be nested within each other:

```
>>> ('David' + 'Mario') * 3
'DavidMarioDavidMarioDavidMario'
```

Set data in Python (`set`)

Python uses the `set` data type to store set data. A `set` literal matches the notation we use in mathematics: the literal begins with a `{` and ends with a `}`, and each element of the list is written inside the braces, separated from each other by commas. For example, `{1, 2, 3}` is a set of `int`s, and `{1, 2.0, 'three'}` is a set of elements of mixed types.

Like other data types, sets can be compared for equality using `==`. Remember that element order does not matter when comparing sets!

```
>>> {1, 2, 3} == {3, 1, 2}
True
```

Python also supports the "element of" ($\in$) set operation using the `in` operator.

```
>>> 1 in {1, 2, 3}
True
>>> 10 in {1, 2, 3}
False
```

Python also allows `not` and `in` to be combined to form an operator that corresponds to the set operation $\notin$:

```
>>> 1 not in {1, 2, 3}
False
>>> 10 not in {1, 2, 3}
True
```

We'll see in the next chapter how other set operations such as union and intersection are supported in Python.

List data in Python (`list`, `tuple`)

Python uses two different data types to store list data: `list` and `tuple`. `list` literals are written the same way as `set` literals, except using square brackets instead of curly braces.

Lists support the same operations we saw for strings and sets earlier:

```
>>> [1, 2, 3] == [1, 2, 3]                          # List equality
        comparison; order matters!
True
>>> [1, 2, 3] == [3, 2, 1]
False
>>> (['David', 'Mario', 'Jacqueline', 'Diane'])[0]  # List indexing
'David'
>>> ['David', 'Mario'] + ['Jacqueline', 'Diane']    # List concatenation
['David', 'Mario', 'Jacqueline', 'Diane']
>>> 1 in [1, 2, 3]                                  # List "element of"
        operation
True
```

`tuple` literals are written using regular parentheses instead, but otherwise support the above operations as well.

```
>>> (1, 2, 3) == (1, 2, 3)                          # Tuple equality comparison
True
>>> (1, 2, 3) == (3, 2, 1)
False
>>> ('David', 'Mario', 'Jacqueline', 'Diane')[0]  # Tuple indexing
'David'
>>> ('David', 'Mario') + ('Jacqueline', 'Diane')  # Tuple concatenation
('David', 'Mario', 'Jacqueline', 'Diane')
>>> 1 in (1, 2, 3)                                  # Tuple "element of"
        operation
True
```

So why does Python have two different data types that represent the same kind of data? There is an important technical distinction between `list` and `tuple` that we'll learn about later in this course, but for now we'll generally stick with `list`.

Mapping data in Python (`dict`)

Python stores mapping data using a data type called `dict`, short for "dictionary". `dict` literals are written similarly to sets, with each key-value pair separated by a colon. For example, we can represent the mapping from the previous section with the dictionary literal `{'fries': 5.99, 'steak': 25.99, 'soup': 8.99}`. In this dictionary, the keys are strings, and the values are `floats`.

But if both sets and dictionaries use curly braces, then does the literal `{}` represent an empty set or an empty dictionary? The answer (for historical reasons) is an empty dictionary—Python has no literal to represent an empty set.[7]

[7] Instead, we represent an empty set with `set()`, which is syntax we haven't yet seen and will explore later.

Dictionaries also support equality comparison using `==`. They support key lookup using the same syntax as string and list indexing:

```
>>> ({'fries': 5.99, 'steak': 25.99, 'soup': 8.99})['fries']
5.99
```

And finally, they support checking whether a key is present in a dictionary using the `in` operator:

```
>>> 'fries' in  {'fries': 5.99, 'steak': 25.99, 'soup': 8.99}
True
```

References

- CSC108 videos: Python as a Calculator (Part 1, Part 2, Part 3)
- CSC108 videos: Type bool (Part 1, Part 2, Part 3, Part 4)
- CSC108 videos: Type str (Part 1, Part 2)
- Appendix A.2 Python Built-In Data Types Reference